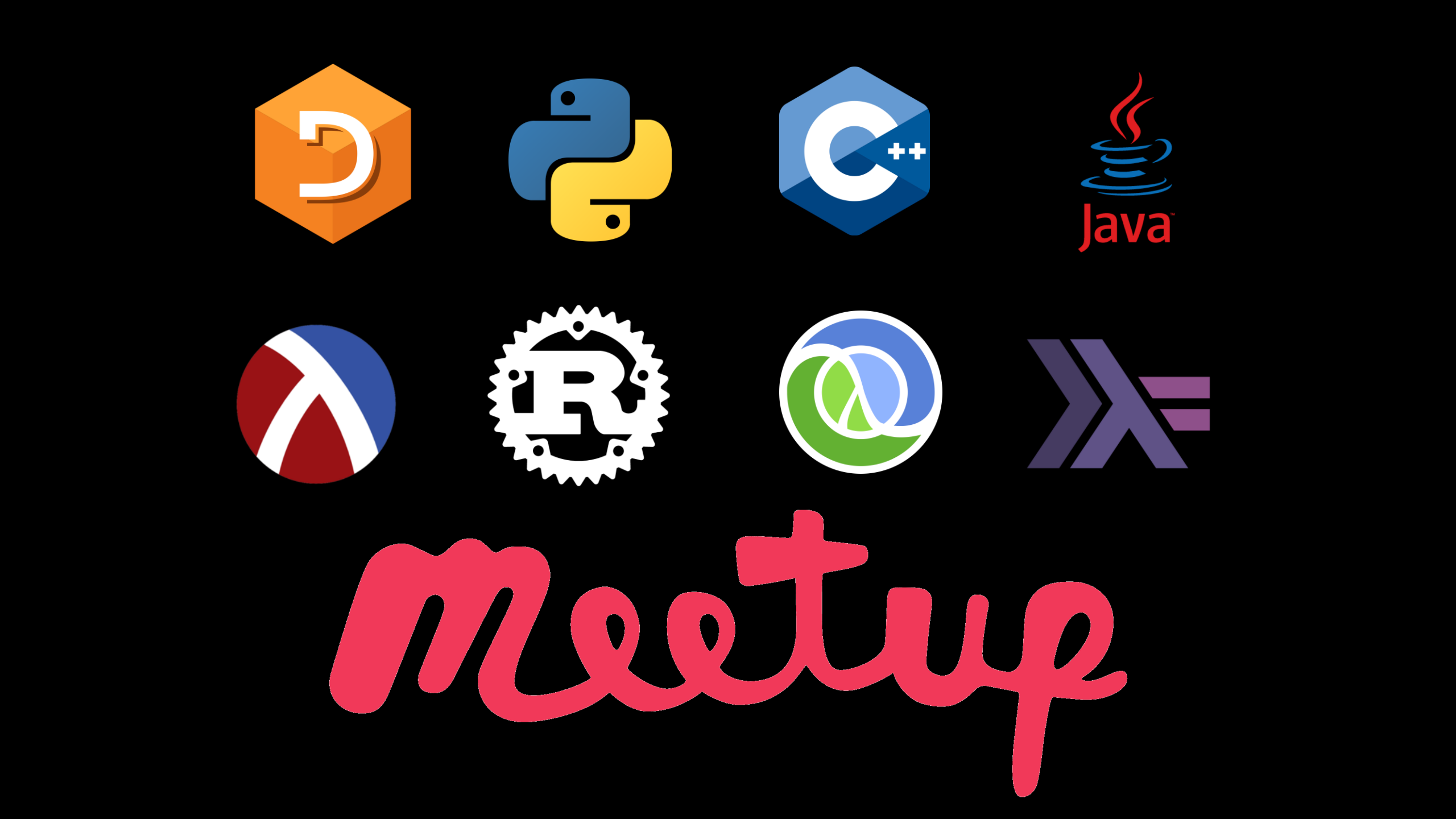
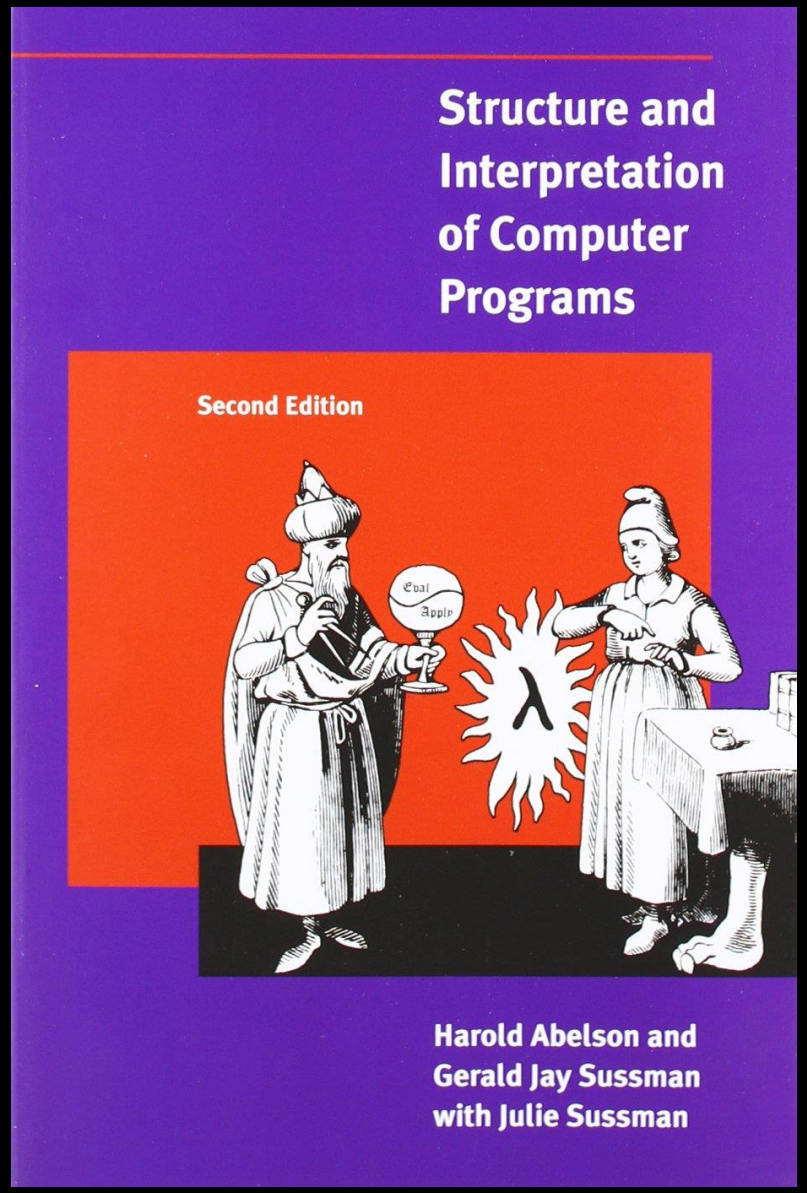

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 5.4

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

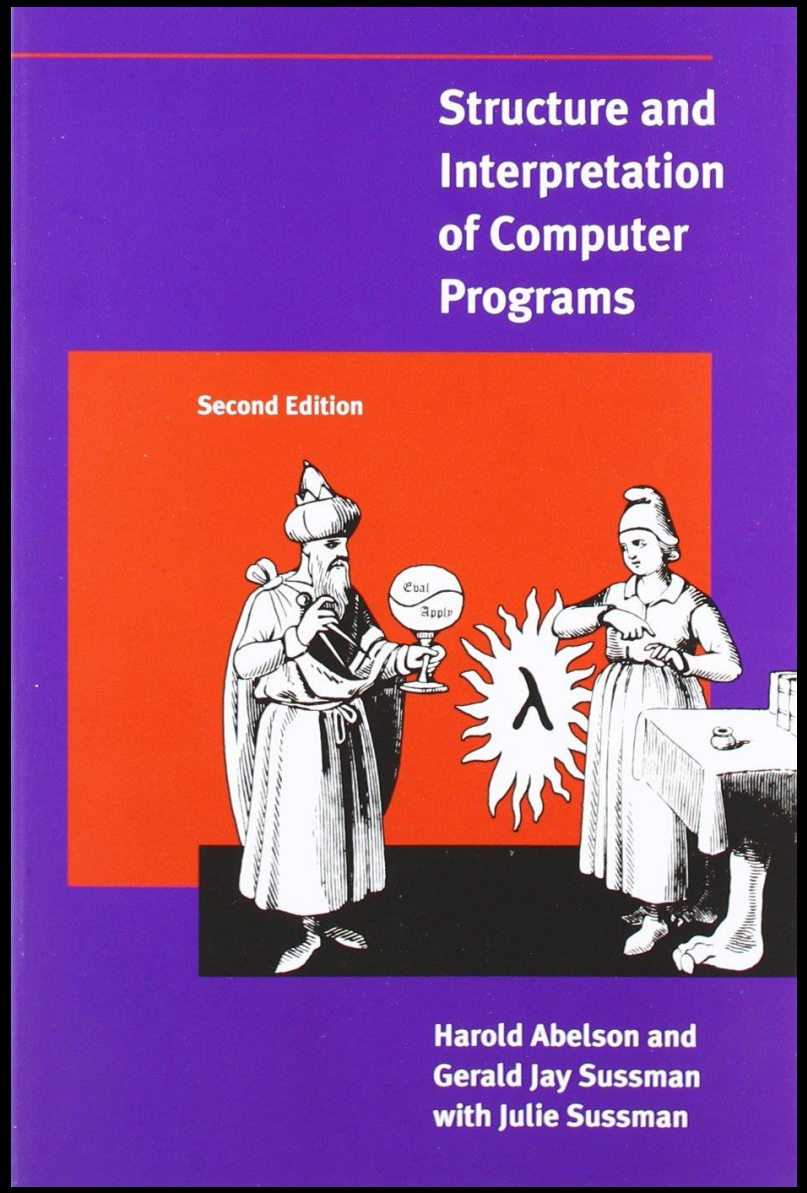
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 5.4

```scheme
(define apply-in-underlying-scheme apply) ; from footnote on page 520

;; https://groups.csail.mit.edu/mac/ftpdir/scheme-7.4/doc-html/scheme_2.html#SEC24

;; A Scheme expression is a construct that returns a value. An expression may be a:
;;    1. literal,
;;    2. a variable reference,
;;    3. a special form,
;;    4. or a procedure call.

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)                              ; #1 Literal
        ((variable?        exp) (lookup-variable-value exp env))  ; #2 Variable Reference
        ((quoted?          exp) (text-of-quotation exp))          ; #3 Special Form
        ((assignment?      exp) (eval-assignment exp env))        ; #3 Special Form
        ((definition?      exp) (eval-definition exp env))        ; #3 Special Form
        ((if?              exp) (eval-if exp env))                ; #3 Special Form
        ((lambda?          exp) (make-procedure                   ; #3 Special Form
                                 (lambda-parameters exp)
                                 (lambda-body exp)
                                 env))
        ((begin?           exp) (eval-sequence                    ; #3 Special Form
                                 (begin-actions exp) env))
        ((cond?            exp) (eval (cond->if exp) env))        ; #3 Special Form
        ((let?             exp) (eval (let->combination exp) env)) ; #3 Special Form
        ((application?     exp) (my-apply (eval (operator exp) env) ; #4 Procedure Call
                                          (list-of-values
                                           (operands exp) env)))
        (else
         (error "Unknown expression type: FAIL" exp))))

(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type: APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

Ch 4.1
~270 LoC

```scheme
;; 5.2.1 The Machine Model

(define (make-machine register-names ops controller-text)
  (let ((machine (make-new-machine)))
    (for-each
     (lambda (register-name)
       ((machine 'allocate-register) register-name))
     register-names)
    ((machine 'install-operations) ops)
    ((machine 'install-instruction-sequence)
     (assemble controller-text machine))
    machine))

;; Registers

(define (make-register name)
  (let ((contents '*unassigned*))
    (define (dispatch message)
      (cond ((eq? message 'get) contents)
            ((eq? message 'set)
             (lambda (value) (set! contents value)))
            (else
             (error "Unknown request: REGISTER" message))))
    dispatch))

(define (get-contents register) (register 'get))
(define (set-contents! register value)
  ((register 'set) value))

;; The stack

(define (make-stack)
  (let ((s '()))
    (define (push x) (set! s (cons x s)))
    (define (pop)
      (if (null? s)
          (error "Empty stack: POP")
          (let ((top (car s)))
            (set! s (cdr s))
            top)))
    (define (initialize)
      (set! s '())
      'done)
    (define (dispatch message)
      (cond ((eq? message 'push) push)
            ((eq? message 'pop) (pop))
            ((eq? message 'initialize) (initialize))
            (else (error "Unknown request: STACK" message))))
    dispatch))

(define (pop stack) (stack 'pop))
(define (push stack value) ((stack 'push) value))
```

Ch 5.2
~350 LoC

```scheme
(define eceval
  (make-machine
   '(exp env val proc argl continue unev)
   eceval-operations
   '(
     ;; 5.4.4 Running the Evaluator
     read-eval-print-loop
     (perform (op initialize-stack))
     (perform
      (op prompt-for-input) (const ";;EC-Eval input:"))
     (assign exp (op read))
     (assign env (op get-global-environment))
     (assign continue (label print-result))
     (goto (label eval-dispatch))
     print-result
     (perform (op announce-output) (const ";;EC-Eval value:"))
     (perform (op user-print) (reg val))
     (goto (label read-eval-print-loop))
     ;;
     unknown-expression-type
     (assign val (const unknown-expression-type-error))
     (goto (label signal-error))
     unknown-procedure-type
     (restore continue) ; clean up stack (from apply-dispatch )
     (assign val (const unknown-procedure-type-error))
     (goto (label signal-error))
     signal-error
     (perform (op user-print) (reg val))
     (goto (label read-eval-print-loop))
     ;; 5.4.1 The Core of the Explicit-Control Evaluator
     eval-dispatch
     (test (op self-evaluating?) (reg exp))
     (branch (label ev-self-eval))
     (test (op variable?) (reg exp))
     (branch (label ev-variable))
     (test (op quoted?) (reg exp))
     (branch (label ev-quoted))
     (test (op assignment?) (reg exp))
     (branch (label ev-assignment))
     (test (op definition?) (reg exp))
     (branch (label ev-definition))
     (test (op if?) (reg exp))
     (branch (label ev-if))
     (test (op cond?) (reg exp)) ; added for Exercise 5.23
     (branch (label ev-cond))    ; added for Exercise 5.23
     (test (op lambda?) (reg exp))
     (branch (label ev-lambda))
     (test (op begin?) (reg exp))
     (branch (label ev-begin))
     (test (op application?) (reg exp))
     (branch (label ev-application))
     (goto (label unknown-expression-type))
     ;;
     ;; Evaluating simple expressions
     ev-self-eval
     (assign val (reg exp))
     (goto (reg continue))
     ev-variable
```

Ch 5.4
~230 LoC

```scheme
(define eceval-operations
  (list (list 'self-evaluating? self-evaluating?)
        (list 'variable? variable?)
        (list 'quoted? quoted?)
        (list 'assignment? assignment?)
        (list 'definition? definition?)
        (list 'if? if?)
        (list 'cond? cond?)
        (list 'lambda? lambda?)
        ;(list 'let? let?)
        (list 'begin? begin?)
        (list 'application? application?)
        (list 'lookup-variable-value lookup-variable-value)
        (list 'text-of-quotation text-of-quotation)
        (list 'lambda-parameters lambda-parameters)
        (list 'lambda-body lambda-body)
        (list 'make-procedure make-procedure)
        (list 'operands operands)
        (list 'operator operator)
        (list 'no-operands? no-operands?)
        (list 'first-operand first-operand)
        (list 'rest-operands rest-operands)
        (list 'empty-arglist empty-arglist)
        (list 'adjoin-arg adjoin-arg)
        (list 'last-operand? last-operand?)
        (list 'primitive-procedure? primitive-procedure?)
        (list 'compound-procedure? compound-procedure?)
        ;(list 'compiled-procedure? compiled-procedure?)
        (list 'apply-primitive-procedure apply-primitive-procedure)
        (list 'procedure-parameters procedure-parameters)
        (list 'procedure-body procedure-body)
        (list 'procedure-environment procedure-environment)
        ;(list 'make-compiled-procedure make-compiled-procedure)
        ;(list 'compiled-procedure-entry compiled-procedure-entry)
        ;(list 'compiled-procedure-env compiled-procedure-env)
        (list 'extend-environment extend-environment)
        (list 'begin-actions begin-actions)
        (list 'last-exp? last-exp?)
        (list 'first-exp first-exp)
        (list 'rest-exps rest-exps)
        ;(list 'compile? compile?)
        ;(list 'compile-and-run compile-and-run)
        ;(list 'compile-and-run-exp compile-and-run-exp)
        (list 'if-predicate if-predicate)
        (list 'if-consequent if-consequent)
        (list 'if-alternative if-alternative)
        (list 'false? false?)
        (list 'true? true?)
        (list 'list list)
        (list 'cons cons)
        (list '= =)
        (list '* *)
```

The Internet
~100 LoC

In Section 5.1 we saw how to transform simple Scheme programs into descriptions of register machines. We will now perform this transformation on a more complex program, the metacircular evaluator of Section 4.1.1–Section 4.1.4, which shows how the behavior of a Scheme interpreter can be described in terms of the procedures `eval` and `apply`.

Our Scheme evaluator register machine includes a stack and seven registers: `exp`, `env`, `val`, `continue`, `proc`, `argl`, and `unev`. `exp` is used to hold the expression to be evaluated, and `env` contains the environment in which the evaluation is to be performed. At the end of an evaluation, `val` contains the value obtained by evaluating the expression in the designated environment. The `continue` register is used to implement recursion, as explained in Section 5.1.4. (The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.) The registers `proc`, `argl`, and `unev` are used in evaluating combinations.

5.4.1 The Core of the Explicit-Control Evaluator

The central element in the evaluator is the sequence of instructions beginning at `eval-dispatch`. This corresponds to the `eval` procedure of the metacircular evaluator described in Section 4.1.1. When the controller starts at `eval-dispatch`, it evaluates the expression specified by `exp` in the environment specified by `env`. When evaluation is complete, the controller will go to the entry point stored in `continue`, and the `val` register will hold the value of the expression. As with the metacircular `eval`, the structure of `eval-dispatch` is a case analysis on the syntactic type of the expression to be evaluated.[20]

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)                              ; #1 Literal
        ((variable?        exp) (lookup-variable-value exp env))  ; #2 Variable Reference
        ((quoted?          exp) (text-of-quotation exp))          ; #3 Special Form
        ((assignment?      exp) (eval-assignment exp env))        ; #3 Special Form
        ((definition?      exp) (eval-definition exp env))        ; #3 Special Form
        ((if?              exp) (eval-if exp env))                ; #3 Special Form
        ((lambda?          exp) (make-procedure                   ; #3 Special Form
                                 (lambda-parameters exp)
                                 (lambda-body exp)
                                 env))
        ((begin?           exp) (eval-sequence                    ; #3 Special Form
                                 (begin-actions exp) env))
        ((cond?            exp) (eval (cond->if exp) env))        ; #3 Special Form
        ((and?             exp) (eval-and exp env))               ; #3 Special Form
        ((or?              exp) (eval-or exp env))                ; #3 Special Form
        ((let?             exp) (eval (let->combination exp) env)); #3 Special Form
        ((application?     exp) (my-apply (eval (operator exp) env) ; #4 Procedure Call
                                          (list-of-values
                                           (operands exp) env)))
        (else
         (error "Unknown expression type: FAIL" exp))))
```

Evaluating simple expressions

Numbers and strings (which are self-evaluating), variables, quotations, and `lambda` expressions have no subexpressions to be evaluated. For these, the evaluator simply places the correct value in the `val` register and continues execution at the entry point specified by `continue`. Evaluation of simple expressions is performed by the following controller code:

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure) (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

Evaluating procedure applications

A procedure application is specified by a combination containing an operator and operands. The operator is a subexpression whose value is a procedure, and the operands are subexpressions whose values are the arguments to which the procedure should be applied. The metacircular `eval` handles applications by calling itself recursively to evaluate each element of the combination, and then passing the results to `apply`, which performs the actual procedure application. The explicit-control evaluator does the same thing; these recursive calls are implemented by `goto` instructions, together with use of the stack to save registers that will be restored after the recursive call returns. Before each call we will be careful to identify which registers must be saved (because their values will be needed later).[21]

We begin the evaluation of an application by evaluating the operator to produce a procedure, which will later be applied to the evaluated operands. To evaluate the operator, we move it to the `exp` register and go to `eval-dispatch`. The environment in the `env` register is already the correct one in which to evaluate the operator. However, we save `env` because we will need it later to evaluate the operands. We also extract the operands into `unev` and save this on the stack. We set up `continue` so that `eval-dispatch` will resume at `ev-appl-did-operator` after the operator has been evaluated. First, however, we save the old value of `continue`, which tells the controller where to continue after the application.

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

Upon returning from evaluating the operator subexpression, we proceed to evaluate the operands of the combination and to accumulate the resulting arguments in a list, held in `argl`. First we restore the unevaluated operands and the environment. We initialize `argl` to an empty list. Then we assign to the `proc` register the procedure that was produced by evaluating the operator. If there are no operands, we go directly to `apply-dispatch`. Otherwise we save `proc` on the stack and start the argument-evaluation loop:[22]

```
ev-appl-did-operator
  (restore unev)                          ; the operands
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))                 ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

Procedure application

The entry point `apply-dispatch` corresponds to the `apply` procedure of the metacircular evaluator. By the time we get to `apply-dispatch`, the `proc` register contains the procedure to apply and `argl` contains the list of evaluated arguments to which it must be applied. The saved value of `continue` (originally passed to `eval-dispatch` and saved at `ev-application`), which tells where to return with the result of the procedure application, is on the stack. When the application is complete, the controller transfers to the entry point specified by the saved `continue`, with the result of the application in `val`. As with the metacircular `apply`, there are two cases to consider. Either the procedure to be applied is a primitive or it is a compound procedure.

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

```
primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))
```

To apply a compound procedure, we proceed just as with the metacircular evaluator. We construct a frame that binds the procedure's parameters to the arguments, use this frame to extend the environment carried by the procedure, and evaluate in this extended environment the sequence of expressions that forms the body of the procedure. `ev-sequence`, described below in Section 5.4.2, handles the evaluation of the sequence.

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

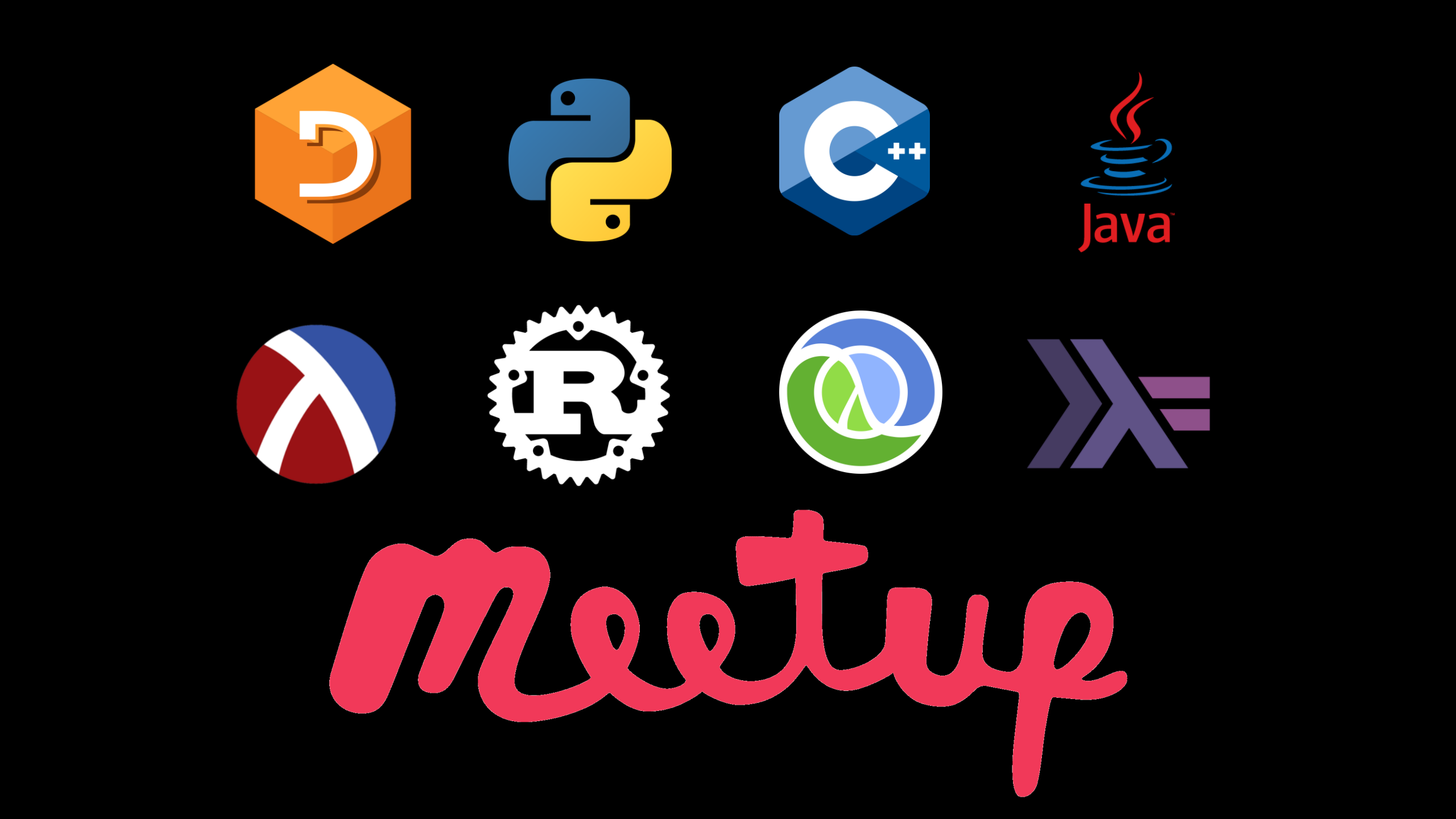
`compound-apply` is the only place in the interpreter where the `env` register is ever assigned a new value. Just as in the metacircular evaluator, the new environment is constructed from the environment carried by the procedure, together with the argument list and the corresponding list of variables to be bound.

Exercise 5.23: Extend the evaluator to handle derived expressions such as `cond`, `let`, and so on (Section 4.1.2). You may "cheat" and assume that the syntax transformers such as `cond->if` are available as machine operations.[28]

```
eval-dispatch
(test (op self-evaluating?) (reg exp))
(branch (label ev-self-eval))
(test (op variable?) (reg exp))
(branch (label ev-variable))
(test (op quoted?) (reg exp))
(branch (label ev-quoted))
(test (op assignment?) (reg exp))
(branch (label ev-assignment))
(test (op definition?) (reg exp))
(branch (label ev-definition))
(test (op if?) (reg exp))
(branch (label ev-if))
(test (op cond?) (reg exp)) ; added for Exercise 5.23
(branch (label ev-cond))    ; added for Exercise 5.23
(test (op lambda?) (reg exp))
(branch (label ev-lambda))
(test (op begin?) (reg exp))
(branch (label ev-begin))
(test (op application?) (reg exp))
(branch (label ev-application))
(goto (label unknown-expression-type))
```

```
eval-dispatch
(test (op self-evaluating?) (reg exp))
(branch (label ev-self-eval))
(test (op variable?) (reg exp))
(branch (label ev-variable))
(test (op quoted?) (reg exp))
(branch (label ev-quoted))
(test (op assignment?) (reg exp))
(branch (label ev-assignment))
(test (op definition?) (reg exp))
(branch (label ev-definition))
(test (op if?) (reg exp))
(branch (label ev-if))
(test (op cond?) (reg exp)) ; added for Exercise 5.23
(branch (label ev-cond))    ; added for Exercise 5.23
(test (op lambda?) (reg exp))
(branch (label ev-lambda))
(test (op begin?) (reg exp))
(branch (label ev-begin))
(test (op application?) (reg exp))
(branch (label ev-application))
(goto (label unknown-expression-type))
```

```
;; Added for Exercise 5.23
ev-cond
(assign exp (op cond->if) (reg exp))
(goto (label ev-if))
```


Java
meetup